

Towards Flexible Finite-State-Machine-Based Protocol Composition*

Richard Ekwall Sergio Mena Stefan Pleisch André Schiper

Distributed Systems Laboratory

Swiss Federal Institute of Technology (EPFL), CH-1015 Lausanne

Tel.: +41-21-693-6463, Fax.: +41-21-693-6770

{nilsrichard.ekwall|sergio.mena|stefan.pleisch|andre.schiper}@epfl.ch

Technical Report IC/2004/63

Abstract

Group communication provides primitives that ensure reliable and ordered delivery of messages to a group of destinations. It is an important building block for replicated fault-tolerant applications such as replicated databases. In the past, most group communication systems have been monolithic. Recent group communication systems have been built around components, allowing for more flexibility.

In this paper, we propose a novel approach to the composition of group communication protocols. In this approach, components are modeled as finite state machines communicating via signals. We introduce two building blocks, called adaptor and adaplexor, that facilitate the development and the composition of group communication protocol stacks, and we discuss how isolation can be achieved in this setting. To validate our architectural concepts, we have implemented the proposed group communication architecture in SDL.

*Research supported partially by OFES under contract number 01.0537-1 as part of the IST REMUNE project (number 2001-65002); partially by the EPFL grant “Semantics-Guided Design and Implementation of Group Communication Middleware”; and partially by the Swiss National Science Foundation under grant number 21-67715.02

1 Introduction

1.1 Context

The designer of protocol stacks has the option to design a stack as a monolithic block or to compose it from a set of protocol modules. The second option has obvious advantages: it makes reuse of protocol modules possible.

In this paper we consider the design of protocol stacks for group communication [4]. Group communication provide abstractions for developing replicated, fault-tolerant applications. Until recently, most group communication systems have been built as a monolithic block with the drawback of reduced component reusability [2]. By contrast, reusability allows to solve the growing need for adaptability.

To achieve reusability and adaptability, protocol modules can be composed in a static way, i.e., the connections between modules have to be known at the compilation of the modules. However, this solution lacks flexibility, and may even make the approach impractical. This is the case especially if some protocol module M may potentially be used by several other protocol modules M' unknown to M when M is built. Either a solution is found to handle this case, or M has to be redesigned later.

1.2 Contribution of the paper

In this paper, we present a flexible protocol composition approach, based on finite state machines (FSM), and we identify three additional building blocks that are required for flexible composition: *adaptors*, *adaplexors*, and *isolators*. While the use of adaptors is fairly obvious and has been proposed as a pattern in [7], adaplexors and isolators are less obvious but important.

To validate our composition approach, we have implemented a prototype of a group communication stack. Each layer of the group communication protocol stack is modeled as a finite state machine. These finite state machines communicate by exchanging *signals*. A signal is a notification that a FSM sends to another FSM. From a composition perspective, we found that this approach has considerable advantages over approaches chosen by other composition frameworks. For example, composing FSMs into a group communication protocol stack does not impose any restrictions on the way the different layers of the protocol stack are implemented: the entire group communication stack can be implemented by a single process, by one process per layer, or any number of layers per process.

The implementation of the protocol stack was done using the Specification and Description Language (SDL) [16, 6]. SDL is a programming language that has been designed with composi-

tion in mind and has been standardised by the ITU. We discovered that SDL is a natural choice for implementing group communication protocol stacks, as its concepts and models nicely fit into our composition approach. Using SDL, the development of the group communication protocol stack became straightforward and the possibility for programming errors was thus greatly reduced. As a result we believe that SDL is much better suited for the implementation of group communication protocol stacks than the general-purpose programming languages such as C, C++ or Java, traditionally used for this purpose. The problem is that these languages are lacking composition features, and had thus to be patched with frameworks such as Cactus or Appia [2, 13] that handle the module composition problem.

1.3 Existing approaches

As stated earlier, there are a number of frameworks for protocol composition in general-purpose languages such as C, C++, and Java. Representative frameworks are Cactus [2] and Appia [13], two composition frameworks. Appia is written for Java and Cactus is written for Java, C, and C++. In both frameworks, the composition is based on events. However, as stated in [12], the Appia and Cactus composition design choice is well suited for point-to-point communication protocols (for example when headers are stripped from a packet by the successive layers in a TCP/IP stack), but is not as well suited for group communication due to the complexity of the interaction between different layers.

1.4 Structure of the paper

The rest of the paper is structured as follows: Section 2 briefly overviews group communication. The model based on finite state machines and signal exchanges is presented in Section 3. In Section 4, we show how protocol modules can be composed in this model. We identify the need for interconnection modules, which are presented in Section 5. Section 6 discusses the issue of isolation in signal processing that occurs in a protocol stack. Section 7 overviews relevant related work. Finally, Section 8 concludes the paper.

2 Group Communication

Protocol composition can be applied to any type of communication stack. In this paper, we focus on group communication protocol stacks, which present more complex interactions between the protocol modules than strictly layered stacks (such as TCP/IP for example).

Group communication (GC) provides abstractions for the development of replicated, fault-tolerant applications such as replicated databases or replicated web servers. It specifies primitives that ensure reliable and totally-ordered communication among a group of processes.

Higher-level abstractions such as total ordering of message delivery can be built on top of lower level abstractions, such as reliable multicast. A protocol stack implementing a particular abstraction thus consists of multiple protocols. In the following, we briefly introduce some of these protocols. A more detailed description of these protocols can be found in [9] and [3]. In this paper, we only consider the protocols needed to understand the contribution of the paper. An example atomic multicast protocol stack is depicted in Fig. 1. We say that a protocol *delivers a message* when it passes the message to the application or upper layer.

- *Reliable Point-to-point (Reliable Pt2pt)*. Reliable point-to-point ensures that a message sent from a sender to a destination arrives at the destination, unless the destination has failed.
- *Reliable Multicast (Rmcast)*. Reliable multicast builds on reliable point-to-point and ensures the so-called atomicity property: if one correct process delivers message m , then all correct processes deliver m .
- *Consensus*. Consensus is a protocol that guarantees agreement among the processes of a group. More specifically, every member of the group proposes an initial value and all group members that have not failed and will not fail for a sufficiently long time eventually decide on the same value among the initial values.
- *Atomic Multicast (Amcast)*. Atomic multicast has the same properties as reliable multicast, but in addition also ensures ordered delivery of messages. In other words, if sender p atomically multicasts message m_p and sender q atomically multicasts m_q , then all group members receive m_p before m_q , or vice-versa. In Figure 1 Atomic multicast uses consensus to agree on a common message delivery order among the group members.
- *Group Membership Service (GMS)*. Finally, the group membership service provides the current membership of the group, i.e., it keeps track of the processes currently belonging to the group.

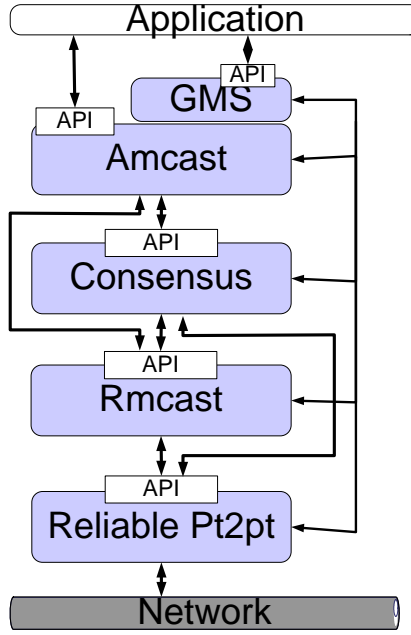


Figure 1. An atomic multicast protocol stack.

3 Model and Definitions

3.1 SDL

SDL (Specification and Description Language) [16, 6] is a widespread, ITU-standardized language in the telecommunications industry. Its primary application field is in specifying communication protocols, but it is suited for any application based on the finite state machine concept, such as circuit design [5]. The programming model used by SDL is based on extended finite state machines (FSM) communicating by exchanging signals, without any shared memory. SDL augments the finite state machine model by providing variables and timers and by supporting object-oriented programming.

3.2 Finite State Machines, Signals and Gates

A *protocol module* implements a particular protocol of the group communication protocol stack. The obvious decision is to model each protocol module as a finite state machine. The finite state machines and thus the protocol modules communicate sending *signals* via *gates*. A signal is a notification that a FSM sends to another FSM. A signal is sent through a gate g_1 of the issuing FSM, to the gate g_2 of the destination FSM. Every FSM has its own thread of control.

Similarly to finite state machines, the SDL designer can set conditions on the signals that are to be received and on the values of their parameters. This makes it easy and natural to express group communication protocols. The protocol implementation is thus close to its pseudo-code description (e.g., [3]), which increases the confidence in the implementation.

We do not need to make any assumptions with respect to the order in which signals are handled and with respect to the time a signal needs to be received by the destination layer, i.e., the time it spends in a gate queue. Indeed, if a gate queue contains many signals waiting to be processed, the time a new signal spends in the signal queue is greater than if the signal queue is empty. However, we assume that signals that are sent via the same gates are received in FIFO order. This is a reasonable assumption, as these signals travel along the same virtual channels.

Moreover, we assume that within a protocol module an incoming signal is processed atomically. In other words, no interleaved signal processing occurs. Hence, the module reads a signal from one of its input gates, processes it, and writes the resulting signal(s), if any, into the corresponding output gate(s). Only then can the processing of the next input signal start.

3.3 SDL system

A SDL system is composed of a set of concurrently-running, finite state machines, that are connected to each other. To facilitate the design and the implementation of large systems, SDL allows the developer to group SDL processes into blocks, which can then be used by higher-level blocks to hierarchically form larger systems. This feature is used for protocol composition in SDL. Each protocol is encapsulated within a block. Blocks are then interconnected to form higher-level protocols, or even a complete system. This approach yields a flexible composition model that can be used for strictly hierarchical stacks (such as a TCP stack for example), and also for stacks with more complex dependencies between the different layers, as is often the case with group communications.

All the implementation examples presented in the following sections are expressed using REMUNE SDL [15], which implements a subset of the original SDL specification.

4 Protocol Composition

The aim of protocol composition is to compose an entire protocol stack from single protocol modules. Traditionally, this has been done in an ad-hoc fashion for each protocol stack. We propose a new approach where this task is achieved by a piece of code that we call *protocol composer*. The protocol composer creates a particular protocol stack out of protocol modules.

The protocol composer also provides the API to the application. This API includes the APIs of all protocol modules that are exposed to the application. Considering the protocol stack in Fig. 1, the protocol composer provides an interface that is identical to the interface of Amcast and GMS.

To allow for the initialization of the protocol stack (via the protocol composer) we have added a *configuration module* with the single method *init*. Signals sent to the *init* method are processed by the *configuration module*. Hence, the group communication stack composer consists of a set of protocol modules and a configuration module. In the following, we present the configuration module in more detail and give an example protocol composer. We then identify a set of generic protocol modules that facilitate the composition of protocol stacks.

4.1 Configuration Module

An important issue in protocol composition is the initialization of the protocol stack. In classical protocol stacks, the protocol modules are responsible to initialize their lower-layer protocol module(s), which, in turn, initialize their lower-layer protocol modules. To provide a common interface to the application and to isolate the composition of the protocol stack from the application, each module has to encapsulate the configuration parameters of the lower-layer protocol modules into its own configuration. However, this creates dependencies between the protocol modules that we want to avoid.

In our composition approach, the initialization of the group communication protocol stack is handled by the configuration module. The configuration module defines the configuration for the particular protocol stack. It parses the configuration parameters and provides initial configuration verification. This verification can detect configuration parameters that may be conflicting between different protocol modules.

4.2 The Protocol Stack Composer

The protocol composer assembles the protocol modules into a working group communication protocol stack. As each protocol module is represented by a FSM with a set of gates and signals, protocol stacks are composed by connecting these FSMs together, i.e., by matching the signals of corresponding FSMs. Figure 2 gives the code for connecting the Amcast protocol module to Consensus and Consensus to Rmcast: *channels* connect two gates.

The three modules (*amcast*, *consensus* and *rmcast*) are first declared. The gate declarations are shown in the following lines. Finally, three channels are declared to connect the blocks together (we assume that the signals of connected gates are compatible). Notice that the

channel declaration (i.e. the connection between two blocks) is done independently of the gate declaration (i.e. the specification of the services needed and provided by a protocol module).

```

BLOCK amcast:AMcast_BT;
BLOCK consensus:Consensus_BT;
BLOCK rmcast:RMcast_BT;

GATE Amcast_Consensus_Gate WITH(Amcast_Consensus_SignalList);
GATE Amcast_Rmcast_Gate WITH(Amcast_Rmcast_SignalList);
GATE Consensus_Rmcast_Gate WITH(Consensus_Rmcast_SignalList);

// Amcast <-> Consensus
CHANNEL
    FROM amcast VIA Amcast_Consensus_Gate TO consen-
sus VIA Consensus_API_Gate;
    FROM consensus VIA Consensus_API_Gate TO am-
cast VIA Amcast_Consensus_Gate;
ENDCHANNEL;

// Consensus <-> Rmcast
CHANNEL
    FROM rmcast VIA Rm_Api_Gate TO consensus VIA Consensus_Rmcast_Gate;
    FROM consensus VIA Consensus_Rmcast_Gate TO rmcast VIA Rm_Api_Gate;
ENDCHANNEL;

// Amcast <-> Rmcast
CHANNEL
    FROM rmcast VIA Rm_Api_Gate TO amcast VIA Amcast_Rmcast_Gate;
    FROM amcast VIA Amcast_Rmcast_Gate TO rmcast VIA Rm_Api_Gate;
ENDCHANNEL;

```

Figure 2. Example code that creates an instance of the Amcast, Consensus, and Rmcast blocks and connects them.

Figure 2 corresponds to the simple case where protocol modules can be easily connected since they have the same interface. In the next section we discuss the problem of connecting protocol modules that have different interfaces. This requires special type of modules, which we call *interconnection module*.

5 Interconnection Modules

In this section, we describe two interconnection modules: *adaptor* and *adaplexor*. We propose the use of these modules as a way to ease the composition of group communication protocol stacks.

5.1 Adaptors

The concept of an adaptor has been proposed as a design pattern in [7]. It allows two protocol modules to communicate although their interfaces may be (slightly) different.

Indeed, these protocol modules may be provided by different vendors and thus use slightly different interfaces and maybe even different specifications. They can thus not be directly connected together. Instead, we use the multiple interfaces approach combined with the *adaptor* design pattern. The adaptor matches the input signals of one module to the output signals of the other module, and vice-versa (see Fig. 3): an adaptor is simply a FSM, i.e. an SDL process encapsulated into a block. The adaptor matching can range from simply connecting corresponding signals (in the case of standard APIs), over syntactical matching based on ontologies, towards semantical matching.

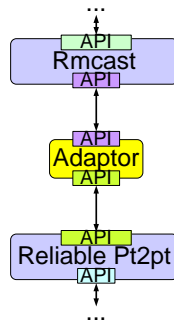


Figure 3. The adaptor matches the APIs of two depending modules.

Consider the common case of an adaptor that only needs to perform minor modifications of the messages sent between two protocol modules. In such a setting, the logic inside the adaptor is often close to trivial, which in turn yields an easy implementation that is not prone to errors. Changing the interface of the protocol implementation directly, on the other hand, might not always be possible, as in the case of a closed-source implementation, or highly impractical because of the large number of interface interactions that need to be modified.

Furthermore, the cost of an adaptor is reasonable. For the stack shown in Figure 1, preliminary performance measurements have shown that the throughput (number of messages delivered by Reliable Multicast per second) decreases by about 7% compared to the same stack without the adaptor.

Finally, in some cases, adaptors can be automatically generated. Assume, for instance, that the corresponding APIs of two modules only differ syntactically. In this case, the developer that

puts the protocol stack together only has to provide a syntactical mapping and then the rest of the adaptor is generated automatically.

5.2 Adaplexors

An adaplexor is a more complex interconnection module. Beside providing the same functionality as the adaptor, it also allows signal de-/multiplexing. We thus propose the name *adaplexor* as a combination of *adaptor* and *multiplexor*. Adaplexors solve the problem that arises when a lower-layer protocol module is used by multiple upper-layer modules (e.g., Reliable Pt2pt or Rmcast in Figure 1).

5.2.1 The Limitation of Hard-Coded Signal Multiplexing and Demultiplexing

The problem can be solved by the developer of such a module, but this requires the developer to be aware of the potential multiple usage of the module (in order to ensure the delivery of signals to the correct upper-layer protocol module). Hence, the module developer would build signal multiplexing and demultiplexing into the module. However, this solution is not ideal, as the module developer has to foresee how the protocol module will be used in future applications! The module should only need to comply with its specification, and not worry about other issues.

Note that another approach is to instantiate such protocol modules multiple times, once for every module that uses it. However, in this case the state of each instance evolves in an uncoordinated manner. This is undesirable most of the time, although there are some particular cases where this may work. Moreover, multiple instances do not solve the demultiplexing problem, rather, the problem is just shifted to the module below. Revisiting the example in Figure 1, assume we instantiate Rmcast twice in order to solve a multiplexing problem at a higher level (e.g., Consensus and Amcast). Connecting both instances of Rmcast to the same instance of Reliable Pt2pt protocol module leads to the same problem at this level. Hence, each instance of Rmcast also needs its own instantiation of Reliable Pt2pt. However, multiple module instances create a large overhead and eventually also result in the allocation of many network resources (e.g., sockets, ports) at the lowest level. Besides, protocols at low levels end up having many instances with no coordination among them.

5.2.2 Flexible Multiplexing using Adaplexors

Adaplexors elegantly address multiplexing and demultiplexing externally to the protocol modules. They multiplex upcall signals, i.e., signals sent from a lower-layer to an upper-layer pro-

protocol module, to the corresponding module (e.g., Amcast or Consensus). Downcall signals, i.e., signals from the upper-layer to the lower-layer protocol module, are demultiplexed to the single lower layer module (e.g., Rmcast). For this purpose, every message originating at a higher-level module is tagged with the name of this module. This tag is then used by the adaplexor to route the signal to the corresponding module in the receiving group communication protocol stack. Figure 4 shows an example adaplexor. An adaplexor is implemented as a FSM, i.e. as an SDL process encapsulated into a block. It offers the same interface as the corresponding lower and upper interfaces it connects.

```

INPUT From_Amcast(Msg);
    HeaderAndMsg := "1" + Msg;
    OUTPUT Rmcast(HeaderAndMsg);
    ...

INPUT From_Consensus(Msg);
    HeaderAndMsg := "2" + Msg;
    OUTPUT Rmcast(HeaderAndMsg);
    ...

INPUT From_Rmcast(Msg);
    Header := String_Left(Msg,1);
    DECISION(Header);
    ("1"):
        OUTPUT To_Amcast(Msg);
    ("2"):
        OUTPUT To_Consensus(Msg);
    ...

```

Figure 4. (Simplified) example implementation of an adaplexor that connects a shared module (Rmcast) to two other modules (Consensus and Amcast). Tags are added to messages to identify the destination module (“1” for Amcast, “2” for Consensus).

5.2.3 Example

Consider the atomic multicast protocol stack in Figure 5. This protocol stack is a composition of the following modules: Consensus, Amcast, Rmcast, and Reliable Pt2pt. Each of these modules is a FSM embedded into a block. The Rmcast and Reliable Pt2pt blocks are used by two other blocks. Instead of using several instances of Rmcast and Reliable Pt2pt, a single instance is combined with an adaplexor to offer the desired functionality. The Consensus and Amcast blocks only know of their own lower interface and are therefore not aware that they are in fact connected to an adaplexor. The adaplexors have the same interface as Rmcast and Reliable Pt2pt; when using the adaplexor for multiplexing, as is the case here, the adaplexor exposes the same interface as the block it is multiplexing.

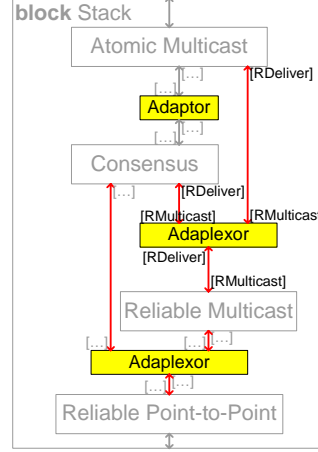


Figure 5. Multiple blocks connected to a single Reliable Broadcast and Reliable Point-to-Point block, using adaplexors

6 Signal Isolation in the Protocol Stack

Composing protocol stacks from finite state machine-based protocol modules leads to interleaved signal processing in the protocol stack. While this may be appropriate for some protocol stack, others require some isolation between the processing of signals. To achieve signal processing isolation within the protocol stack, we propose a novel protocol module, called *isolator*. The isolator relies on well-known algorithms to ensure isolation [1, 8]. The main contribution of this section is the definition of the isolator and mechanisms required in order to allow these algorithms to be applied.

6.1 The Problem of Interleaved Signal Processing

In our model, we assume that signals are processed atomically within a protocol module (see Section 3). However, this is not the case for the entire protocol stack, where different protocol modules can each process a signal concurrently. Indeed, a particular signal may need to be processed by a set of protocol modules before any other signal can be processed by any module in the same set. Consider the example (partial) stack consisting of Reliable Pt2pt and Rmcast [17]. Both modules accept notifications about the current members of the group (called *view change* signals, or *VC*) emanating from the Group Membership Service (GMS) (see Fig. 1). These signals convey information about processes that need to be added or removed from the group. Assume that a new process has joined the group and that GMS has sent a signal to Rmcast,

which has updated its local view to include the new process. Assume further that Reliable Pt2pt has not yet processed that view change signal by the time Rmcast broadcasts a message to all group members in the *new* view. As Reliable Pt2pt has not received the view change signal yet, it does not know the new process, and thus, it will simply drop the message that is to be sent to the new process. This violates the specification of Rmcast and may lead to an incorrect execution. Note that the message is dropped to avoid the possibility of keeping it forever.¹ To prevent this misbehavior, both Rmcast and Reliable Pt2pt must process the view change notification before any other signal is processed by either module. Hence, S_1 followed by S'_1 or S_2 followed by S'_2 need to be processed either before or after both signals VC_1 and VC_2 (see Fig. 6). We say that S_1 and S'_1 , S_2 and S'_2 , VC_1 and VC_2 run in isolation, respectively [8]. Consequently, we exclude, for instance, the following sequence of signal processing: S_1, VC_2, S'_1, VC_1 .

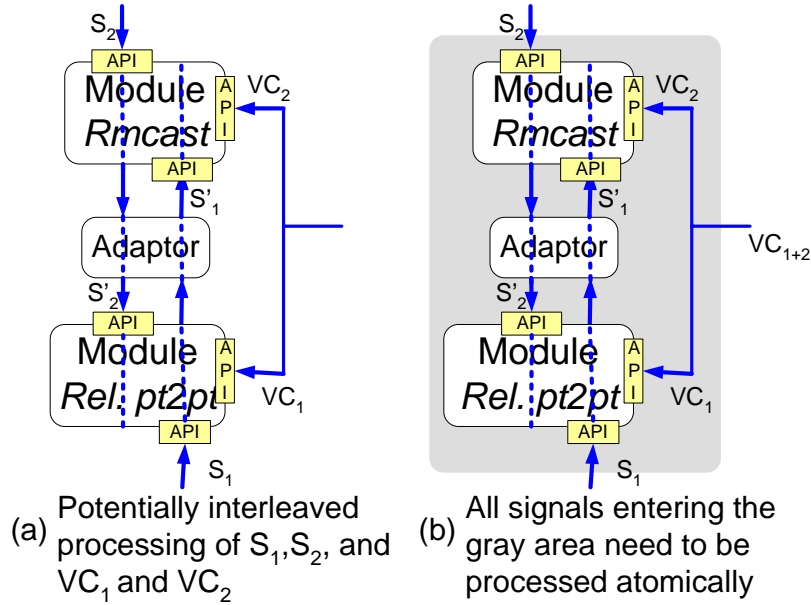


Figure 6. Messages entering the gray box need to be dealt with atomically, i.e., as if they execute in isolation.

Note that isolation is trivially achieved among signals that are sent through the same sequence of gates within the protocol stack. In this case the signals are implicitly ordered by the FIFO property at the gates and the atomicity of signal processing within a protocol module.

¹This occurs when a process unknown to Reliable Pt2pt is from a *past* view rather than from a *future* view.

6.2 The Isolator Module

A protocol stack thus needs to provide a mechanism that enforces isolation in the signal processing. The most stringent mechanism only allows a single signal to be processed at any time in both reliable point-to-point and reliable multicast. Revisiting the example in Fig. 6(a), only one single signal can be processed within the gray area (see Fig. 6(b)). In other words, the signals are processed in a strictly sequential order (here, we consider VC_1 and VC_2 as *one* signal). Clearly, the strictly sequential order is not needed in all cases, as it results in reduced performance. Rather, signal processing can be interleaved as long as the outcome corresponds to the result of some sequential processing [8, 17].

It is instrumental for any isolation mechanism to know when the signal processing in a protocol module has terminated. Earlier approaches have thus built the isolation mechanisms into the runtime environment, where they have access to the execution threads that process the signals [17]. In our model (FSM with signal exchange), we do not have access to the runtime system. As a consequence, isolation needs to be ensured based only on the information gained from the input signals to and output signals from the protocol modules. We thus propose a novel protocol module, called *isolator*, that is added to the protocol stack. Fig. 7 shows the isolator for the Rmcast and Reliable Pt2pt modules. All input and output signals to and from these two protocol modules are routed through the isolator. Hence, the isolator can decide when to forward a particular input signal to the modules. By controlling the module's input and output signal, the isolator enforces signal processing isolation within a protocol stack. For this purpose, it can use any of the isolation approaches to general transaction processing proposed in [8] and also in [17].

However, it is not always possible to clearly determine when an input signal processing is terminated. Some protocol modules generate an output signal for every input signal they receive. In this case, the isolator knows exactly when the handling of the signal within a protocol module is done. However, in some protocol modules an input signal not always generates an output signal, or may generate multiple output signals. Indeed, assume that protocol module Rmcast *consumes* some signal, i.e., an input signal does not trigger a corresponding output signal. If this occurs from time to time, the isolator cannot know when the processing of an input signal is terminated and thus when to process the next signal. This is a consequence of the assumption that no bounds can be placed on the time a signal spends in the signal queue. Hence, isolation cannot be ensured without requiring such modules to explicitly notify the termination of the input signal processing.

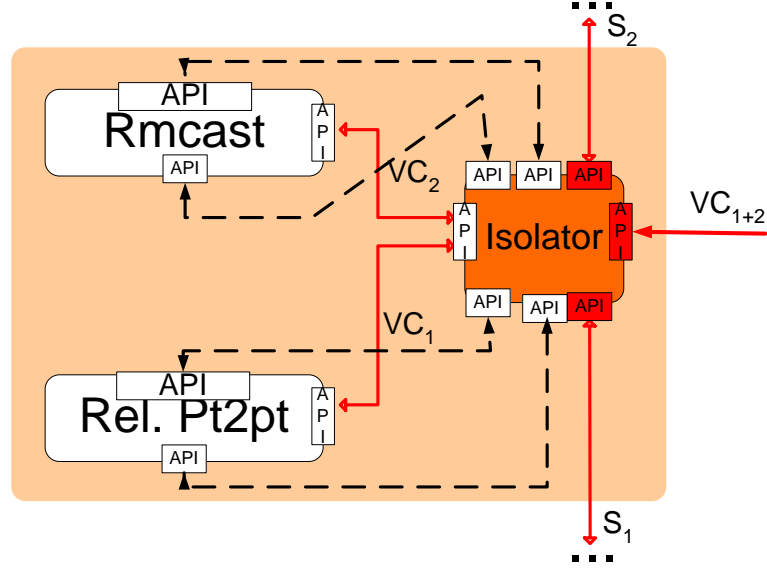


Figure 7. The isolator module for the Rmcast and Reliable Pt2pt modules. The lines denote view change signals, while dotted lines denote normal signals. For simplicity, we assume that no adaptor is needed.

6.3 Termination-Notified Signal Processing

We denote the processing of a signal by *termination-notified* signal processing if a particular signal, called *termination* signal, is generated in the protocol stack to indicate the completion of signal processing. The termination signal corresponding to signal S_1 is called $\overline{S_1}$.

It is easy to see why termination-notified signal processing allows the isolator to ensure isolation. Indeed, the isolator controls all signals that are sent to and from a protocol module that is relevant for isolation (see Fig. 7). When it forwards a signal to a particular protocol module, it waits until it receives the corresponding termination signal and then, based on the type of signals and the isolation constraints, decides upon the next signal(s) to forward.

The following two approaches achieve termination-notified signal processing: (1) positive termination signal or (2) negative termination signal. Both approaches require the collaboration of the protocol modules and thus impose restraints on the developers of these modules. Protocol modules that do not comply with these requirements cannot be integrated into a protocol stack that supports isolation. This limits to a certain extent the applicability of third-party protocol modules and the flexibility of the protocol stack.

6.3.1 Positive Termination Signal

In the *positive termination signal* approach, every module explicitly indicates the termination of the processing of a signal. For this purpose, every module contains a so-called *isolation gate*, via which it sends the termination signal. The termination signal contains the ID of the corresponding input signal, and a list of output signal IDs. The latter are the result of processing the input signal.

The advantage of this approach is that the isolator has complete knowledge about the signal processing in the protocol stack. However, this approach has a high overhead, as for every module and every input signal, a corresponding termination signal is generated.

Clearly, the positive termination signal approach trivially ensures that the isolator knows the exact moment the processing of an input signal is terminated. From the termination signal it learns the number of output signals corresponding to a particular input signal and can use this information, if needed in the isolation mechanism, to wait for the reception of these output signals.

6.3.2 Negative Termination Signal

In the *negative termination signal* approach, a termination signal is only generated in components that potentially consume a signal. Hence, each module specifies a so-called *normal* behavior, i.e., it indicates how many output signals are usually generated from a corresponding input signal. If in some particular cases, not as many signals are generated, then a termination message is generated instead of the missing output signal(s).

Using this approach, the isolator can precisely determine when the processing of a signal by a module is terminated. Indeed, either the isolator receives all the signals specified by the normal behavior of the protocol module, or it receives a termination signal indicating which output signals will not arrive. Clearly, this approach is not applicable if more signals are generated than in the so-called normal behavior. In such a case, the normal behavior should be defined as corresponding to the highest number of output signals.

6.4 Implementation

Similarly to the adaptor and adaplexor, an isolator is also implemented as a FSM. In SDL, it is thus represented by a block.

Note that the developer of the protocol composer can decide on how many FSMs to use for the interconnection modules and the isolator modules in the protocol stack. On one hand, he could implement all of these modules by a separate FSM per module. On the other hand, he could

use a single FSM that implements all these modules. Clearly, the latter case results in a more complex module, but has the advantage of adding only one additional FSM. Moreover, it allows for a certain degree of flow control and prioritizing of signals.

7 Related Work

Protocol composition using general-purpose languages such as Java or C is supported by a number of frameworks. These frameworks offer abstractions to bridge the gap between the group communication system model (message exchange) and the functionality offered by general-purpose languages (function calls).

The Appia framework [13] reuses and extends the protocol composition framework designed for Ensemble [10, 11]. Appia and Ensemble provide hierarchical protocol composition. Protocols interact by means of events, which play the role of signals in these frameworks. Events traverse a number of protocols following a route defined at event creation time. In contrast, as we have shown in previous sections, group communication protocols use mainly point-to-point events, i.e., events that do not traverse any protocol: they are created at a protocol and disposed of at the first protocol they reach. As a result, group communication protocols implemented in Appia do not profit from its complex mechanisms that route events across several layers.

An important feature of Appia is the possibility to have several possible predefined routes for events, called *channels*. Indeed, having several Appia channels allows event multiplexing in a similar way as *adaplexors* proposed here. However, this channel-based technique is not as flexible as *adaplexors*, since an event sets its channel (i.e., its route) at the time it is first created (or injected from the network) and this route cannot be changed later on.

Cactus [18], extends the *x*-kernel [14] protocol framework to allow for a finer-grain level of composition. In Cactus, the internal structure of an *x*-kernel protocol consists of the composition of several protocols (called *micro-protocols* in [18]). Similar to our model, these protocols are event-driven² and their composition is not hierarchical, allowing them to directly interact without artificial restrictions imposed by protocol stack hierarchy. Cactus allows several event handlers to be bound to the same event so that all these handlers are executed upon occurrence of this event. Although this interaction pattern is simpler than that provided by Appia, it is still more complex than the one needed by group communication protocols, in which (point-to-point) events do not need to execute more than one handler.

In the paper, we have presented some important compositional problems and proposed *adaptors* and *adaplexors* as solutions for them. Appia and Cactus do not provide solutions for these

²*Events* in Cactus are similar to *events* in Appia and *signals* described in this paper.

compositional problems. Hence, when developers of protocol stacks are confronted to these compositional problems, they need to implement adaptors or adaplexors as ad-hoc protocols. This was the case in the group communication stack we implemented in Appia and in Cactus [12].

The problem of isolation in the context of transactions is well-studied [1, 8]. However, a transaction starts with a begin transaction and ends with an end transaction. Consequently, the scope of the transaction is well-known and its termination is a clearly specified event (the end transaction command). In our composition, however, this is not the case. In Appia and Ensemble, isolation is trivially achieved. Only one thread executes all events so that at most one event handler is executed among all protocols at a given time. This thread executes an event and all its resulting events to completion before executing the following event that comes from the network/application. Appia and Ensemble provide also other execution order guarantees like FIFO: given two events e and e' flowing in the same direction through a channel, if protocol p handles e before e' then no protocol will handle e' before e .

Cactus does not bound the level of concurrency. When a protocol needs to trigger a new event, there are two ways to do it. Synchronous event triggering (called *invoke*) blocks until the framework has finished executing all event handlers bound to the triggered event. Asynchronous event triggering (called *raise*) returns immediately after the call, and thus executes in concurrence with the handlers bound to the triggered event. Synchronous triggering (event invocation) does not increase concurrency. In the case of asynchronous triggering (event raising), a concurrent computation is spawned. However, the framework does not provide any means to control this concurrent execution of handlers. In the end, it is up to the protocol developers to implement concurrency control by means of standard operating system synchronization mechanisms (locks, semaphores, monitors, etc).

Isolation in the context of group communication stacks has been discussed in [17], where an extension to the Java programming language is proposed. The paper assumes that the end of an event execution is explicitly known, i.e., by having access to all signal queues. In SDL, we do not assume access to the signal queues, and thus our approach is more general. We do, however, rely on the techniques proposed in [17] and [1, 8] to achieve isolation; the specific problem addressed here is to determine the termination of the event processing.

8 Conclusion

In this paper, we have presented a flexible composition approach for communication protocols. In this approach, components are modeled as finite state machines communicating via signals. In order to improve composition flexibility, we have introduced two interconnection modules:

adaptors and *adaplexors*. These modules ease protocol development and composition in a number of ways. *Adaptors* match module interfaces that otherwise would not be compatible, even if they provide the same functionality. *Adaplexors*, apart from featuring adaptors functionality, allow several modules to interact with the same lower-level module that was maybe designed without this multiplicity in mind.

Group communication protocol stacks, and protocol stacks in general, may require that particular signals are processed in isolation. We showed how isolation can be achieved in this setting. For this purpose, we have introduced a new module, called *isolator*.

We have validated our architectural concepts by implementing a prototype group communication stack. Our language of choice has been SDL, whose concepts and system model naturally fit our composition approach. As a result, we believe SDL is much better suited for the implementation of group communication protocol stacks than general-purpose languages (such as C, C++, or Java), which need to be augmented with composition frameworks in order to provide the same support for protocol composition.

In the future, we plan to evaluate the performance of our group communication stack. In particular, we are interested in measuring the overhead introduced by the interconnection modules and the isolator.

References

- [1] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Massachusetts, USA, 1987.
- [2] N.T. Bhatti and R.D. Schlichting. A system for constructing configurable high-level protocols. In *SIGCOMM*, pages 138–150, 1995.
- [3] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J ACM*, 43(2):225–267, March 1996.
- [4] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):427–469, May 2001.
- [5] G. Csopaki and K. Turner. Modelling digital logic in sdl. In *Proc. of the Joint Int. Conference on Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE/PSTV'97)*, pages 367–382, 1997.
- [6] J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL: Formal Object-Oriented Language For Communicating Systems*. Prentice Hall, Harlow, England, 1997.
- [7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Reading, Massachusetts, USA, 1995.

- [8] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, USA, 1993.
- [9] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. Technical Report 94-1425, Department of Computer Science, Cornell University, May 1994.
- [10] M. Hayden. The Ensemble system. Technical Report TR98-1662, Department of Computer Science, Cornell University, January 8, 1998.
- [11] J. Hickey, N. Lynch, and R. van Renesse. Specifications and Proofs for Ensemble layers. In R. Cleaveland, editor, *5th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *LNCS*, pages 119–133. Springer Verlag, March 1999.
- [12] S. Mena, X. Cuvellier, C. Grégoire, and A. Schiper. Appia vs. cactus. In *Proc. of 22th IEEE Symposium on Reliable Distributed Systems (SRDS'03)*, Florence, Italy, October 2003.
- [13] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of The 21st Int'l Conf. on Distributed Computing Systems (ICDCS-21)*, pages 707–710, Phoenix, Arizona, USA, April 2001. IEEE Computer Society.
- [14] L. Peterson, N. Hutchinson, S. O'Malley, and H. Rao. The X-kernel: A platform for accessing Internet resources. *Computer*, 23(5):23–33, May 1990.
- [15] IST/IMS Project. Advanced real-time multi-media and networking execution platform and development environment (REMUNE). <http://lsrwww.epfl.ch/Research/Remune/>.
- [16] SDL Forum Society. SDL specification (z.100 11/99). <http://www.sdl-forum.org>.
- [17] P. Wojciechowski, O. Rütti, and A. Schiper. SAMOA: Framework for synchronization augmented microprotocol approach. In *Proc. of Int. Parallel and Distributed Processing Symposium (IPDPS'04)*, Santa Fee, US, 2004.
- [18] G.T. Wong, M.A. Hiltunen, and R.D. Schlichting. A configurable and extensible transport protocol, April 2001.